The Design and Implementation of the Intel® Real-Time Performance Analyzer

Leena K. Puthiyedath, Erik Cota-Robles, John Keys, Anil Aggarwal, Jim P. Held
Intel Labs
Leena.K.Puthiyedath@intel.com, Erik.Cota-Robles@intel.com, Jim.P.Held@intel.com

Abstract

Modern PCs support growing numbers of concurrently active independently authored real-time software applications and device drivers. The non real-time nature of PC OSes (Linux, Microsoft* Windows*, etc.) means that robust real-time software must cope with hold-offs without degradation in user perceivable application Quality of Service. The open nature of the PC platform necessitates measuring OS interrupt and thread latencies under concurrent load in order to determine with how much hold-off the application must cope.*

The Intel® Real-Time Performance Analyzer is a toolkit for PCs running Microsoft Windows. The toolkit statistically characterizes thread and interrupt latencies plus Windows Deferred Procedure Call (DPC) and kernel Work Item latencies. The toolkit also has facilities for analyzing the causes of long latencies. These latencies can then be incorporated as additional blocking times in a real-time schedulability analysis. An isochronous workload tool is included to model thread and DPC based computation and detect missed deadlines.

1. Introduction

In recent years real-time and in particular multimedia and soft device software have become widespread on desktop personal computers (PCs) and workstations. Traditionally real-time systems have been closed box systems that ran specialized RTOSes with deterministic service times. In contrast, PC and workstation systems are open boxes and the vast majority of them run general purpose OSes.

As a result the execution environment on desktop PC and workstation platforms is highly dynamic and will generally include a number of concurrently executing real-time and non-real-time applications. Too, the general purpose OSes used on desktop PCs and workstations, including Linux and the many variants of Microsoft Windows, do not provide deterministic service times for either response to external events or for OS provided services. As a result real-time software on desktop PCs must be designed to cope with long hold-offs, or latencies, for OS services such as the dispatch of a kernel mode thread. The Intel Real-Time Performance Analyzer characterizes the latencies that a PC application running on top of Microsoft Windows should expect (i.e., which it should be designed to withstand).

By *latency* we mean the duration from a causal event to a dependent event, so that *interrupt latency*, for example, is the duration between when the interrupt "occurs" and when the software ISR actually starts executing. By *thread latency* we mean the delay between when a thread becomes ready to run (e.g., is unblocked) and when it is actually dispatched. In previous work [5] we showed that the magnitude and frequency of these latencies are highly sensitive to the amount of OS overhead that is incurred to service concurrently executing applications.

The Intel Real-Time Performance Analyzer automates the collection of these and additional Windows-specific real-time performance metrics as well as the post analysis of data. By using the Analyzer an engineer can quickly characterize the real-time Quality of Service (QoS) that his/her application will receive under load. In addition, the Analyzer contains workload tools that enable dynamic simulation of various design alternatives and rapid assessment of their robustness under a variety of concurrent load conditions likely to be encountered on an open platform such as the PC.

A key feature of these types of latency performance metrics is that to yield meaningful data, measurements must be made under load from concurrent applications. In general the measured distributions will not be normal, with the result that many of the well-known tools of elementary statistics do not apply. In particular, the mean, standard deviation and variance are largely irrelevant. To address this the Analyzer contains post-processing tools that automatically display collected data in a statistically meaningful way.

The remainder of this section provides background on the Windows NT 4.0 and WDM driver models and surveys prior work on OS performance analysis tools. Section 2 describes the design of the Analyzer while

* Other names and brands may be claimed as the property of others.

Published in the *Proceeding of the Eighth Real-Time and Embedded Technology and Applications Symposium*, Sept. 2002

section 3 describes key aspects of implementation. Section 4 presents two case histories of analysis performed with the tool, section 5 presents lessons learned and section 6 concludes.

1.1 Background

For those unfamiliar with Windows NT 4.0, Windows 2000 or Windows XP, a few definitions may be helpful:

- **APC**: Asynchronous Procedure Call. A means for threads to schedule a callback at elevated IRQL (i.e., above the thread priority space).
- **APC_LEVEL**: 2nd lowest IRQL.
- **Critical Work Item**: a kernel mode thread that runs at default real-time priority services **Work Item**s in this queue.
- **DPC**: Deferred Procedure Call. In Windows an **ISR** can queue a **DPC** to a FIFO queue to do time-critical work on its behalf. **DPC**s execute after all **ISR**s but before threads. This is similar to the Immediate queue for **ISR** "Bottom Halves" in Linux*, but DPCs can be executed on any processor in the manner of Linux "tasklets".
- **DISPATCH_LEVEL**: 3rd lowest IRQL scheduler space. All DPCs run at this level and it is also known as DPC_LEVEL.
- **DRQL**: Device **IRQL**. Elevated range of IRQLs for device driver ISRs (above DPC_LEVEL).
- **HAL**: Hardware Abstraction Layer. Windows provides a processor abstraction so that the same driver can run on all supported processors.
- **HIGH_LEVEL**: Above DRQL, so device interrupts are held off from being serviced.
- **IOCTL**: Device IO ConTroL code passed to the Win32 DeviceIoControl interface to specify a device-specific (control) operation.
- **IRQL**: Nominally, Interrupt ReQuest Level. Windows has a hierarchical scheduler with the IRQL levels as root scheduler and the thread scheduler as a leaf scheduler. The IRQL levels are a static priority preemptible scheduler with priority inheritance and use the Highest Locker protocol to bound priority inversions.
- **ISR**: Interrupt Service Routine. Typically in Windows ISRs queue **DPC**s to transfer data.
- **PASSIVE_LEVEL**: Lowest IRQL, nominally all threads run at this level, but see also **APC**.
- **PIT**: Programmable Interval Timer. PC hardware timer. By default on Windows it fires at a frequency of 67--100 Hz (10--15 ms. period).
- **Real-time Priority**: The Windows NT 4.0, 2000 and XP thread schedulers have 16 real-time priorities, 16 through 31. 24 is the default. Most, but not all, user mode threads execute at non-real-time priority and threads set to a real-time priority preempt non-real-time threads.
- **Thread Scheduler**: Windows threads are owned by processes and categorized as "user mode" and "kernel mode". Both can have real-time priorities. Threads are scheduled preemptively according to their priorities. Priorities of threads in the real-time priority range are static while priorities in the non-real-time range are dynamic.
- **Work Item**: The kernel maintains several "Work Item" queues for callback functions. Kernel mode threads service the queues and **DPC**s use Work Items to do work at **PASSIVE_LEVEL**.

At some level of abstraction the Windows Driver Model (WDM) [3][15][18] is quite similar to the Unix driver model. Drivers expose load and unload, and create and close interfaces to respectively enable the OS to make a device available and an application to access a file on the device. Data and control signals are exchanged using read, write and io_control interfaces. All of these interfaces use IO Request Packets (IRPs) for communication with the driver. ISRs for WDM device drivers typically perform only a limited amount of time critical work and queue a DPC to transfer data to/from the device. WDM provides a layered driver architecture not unlike Unix streams and a processor abstraction layer (the HAL). WDM also includes facilities for auto detection of devices using "Plug and Play" as well as many other features that need not concern us here.

1.2 Previous Work

Prior work on OS benchmarking and performance analysis tools can be divided into two categories: general purpose and real-time performance analysis. General-purpose benchmarking tools include batch macrobenchmarks and microbenchmark suites. Batch macrobenchmarks are widely available and range from the compute-intensive such as SpecInt* to the wholly application-oriented such as Winstone* [20]. They typically drive the system as quickly as possible and produce one or a few numbers for system throughput that correlate well to overall system performance but fail utterly to correlate with real-time performance [5].

Microbenchmarks, in contrast, measure the average cost of low-level primitive OS services over thousands of invocations in a tight loop. The limited resolution of traditional hardware timers originally forced these tools to measure cost in this manner. In the literature these measurements have generally been performed on unloaded systems in order to isolate OS effects from hardware effects (e.g., warm caches), with the result, as Bershad et. al. note [1], that microbenchmarks have not been very useful in assessing the overhead that an application or driver will actually receive from the OS.

There are two main microbenchmark suites in use today. *Lmbench*[*], by McVoy and Staelin, extends work by Ousterhout [16], and is a portable suite for measuring OS as well as hardware performance primitives [11]. Brown and Seltzer in turn extend lmbench to create *hbench:OS*[*], which utilizes the performance counters on the Intel® Pentium® and Pentium Pro processors to instrument the OS [2]. Although the use of the processor performance counters should enable hbench:OS to accurately characterize individual OS events, Brown and Seltzer revise the lmbench measurement of context switch time to exclude any effects from cache conflict. The claim is that this enables hbench:OS to produce measurements with a much smaller standard deviation and that cache miss costs can be added in separately. While true in theory, in practice the relationship of process switching, external load and cache miss rate can be hard to quantify exactly.

Although both benchmarks measure context switch time neither lmbench nor hbench:OS directly measures response to interrupts. Although Endo, et. al. do measure response to keyboard and mouse interrupts but their focus is on interactive performance [6]. In contrast, real-time benchmarks such as Rhealstone[*] [10] do measure interrupt response time as well as context switch time. In the real-time literature measurements have typically focused on worst case cost on a system that is either unloaded or has only a single application. While adequate for dedicated real-time systems this approach is overly pessimistic for general purpose OSes in open environments [4].

Another set of real-time tools are the real-time task simulation tools such as Hartstone[*] [13], Dynbench[*] [17] and DRTSS[*] [19]. These enable the user to model task sets and experiment with varying periods, amounts of intertask synchronization, etc. to determine the extent of the design space where deadlines can be satisfied. Although these tools enable one to determine the effects of the total system on the modeled application, they do not allow one to determine the effects of the modeled application on the system's overall ability to deliver adequate real-time Quality of Service (QoS) to other applications. In other words, although these tools can map out the design space where a modeled application receives good QoS, they cannot map out the design space where a modeled application is a "good citizen". On open environments such as the PC the latter can be as important as the former, since a "bad citizen" application will break other applications and drivers.

Finally, although not strictly a tool, Jones and Regher describe a methodology of instrumenting the Windows NT kernel and making latency measurements [9]. Besides requiring source code to the OS this technique, as the authors note, can be quite tedious in practice, requiring multiple iterations of instrument--rebuild the OS--measure--analyze in order to pin down a problem.

2. Tool Design and Features

The Intel Real-Time Performance Analyzer toolset has three main features:

1. Statistical analysis of scheduling latencies at different priority levels
2. System analysis of high priority scheduling events that occurred during a sampling interval
3. An isochronous workload simulator

The statistical and system analysis features are implemented with a WDM driver for data collection ("data collector driver") and a Graphical User Interface (GUI) to post process the data and graph and log the data. The isochronous workload simulator tool is implemented as a separate WDM driver for scheduling the periodic tasks and a simple GUI to configure and display if the computation met its periodic deadlines.

Both the data collector driver and isochronous workload driver use Pentium processor resources, specifically the local Advanced Programmable Interrupt Controller (APIC), Performance Counters and Time Stamp Counter [7]. The APIC and Performance Counters are configurable resources and inasmuch as there are no standard interfaces in the Windows OS to manage the configuration of these resources when shared, the analyzer implementation provides two WDM drivers that provide sharing interfaces: an APIC driver and a Performance Counter driver. These drivers assume they have sole ownership of these processor resources and the analyzer features that depend on these drivers will not be enabled if any other service in the system uses the local APIC or processor performance counters. The data collector driver and the isochronous workload drivers access the Pentium Time Stamp Counter (TSC) directly for timing information. These timestamp measurements are accurate to the processor clock cycle. The GUI optionally converts the clock cycles to microseconds to ease analysis.

The data collector driver creates a system thread to measure thread latencies. Because the thread does not create an APC all measurements are for a thread executing at IRQL PASSIVE_LEVEL (see section 1.1 for definitions). The driver provides Device Control IOCTLs for the GUI application to control statistical and system analysis sampling and stores data points in a large circular buffer for later retrieval by the GUI application.

2.1 Statistical Latency

Interrupt latency is the delay from the assertion of a hardware interrupt, *as seen by the processor*, until the first instruction of the software interrupt service routine (ISR) is executed. Thus, it measures the total delay to initial servicing of an interrupt. This encompasses any time during which interrupts are disabled as well as the bus

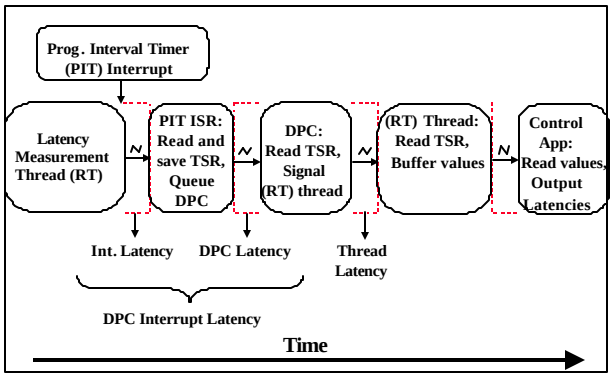

Figure 1: WDM Interrupt, DPC and Thread Latency Measurement

latency necessary to resolve the interrupt, but does not include the bus latency prior to the assertion of the interrupt at the processor. Figure 1 depicts an idealized timeline with interrupt latency, DPC latency, thread latency and thread context switch time marked.

Interrupt latency is measured for the Programmable Interrupt Timer (PIT). The driver supplies its own timer ISR that records the processor Time Stamp Counter (TSC) every time the Timer ISR is executed and then jumps to the system's timer ISR. Since the driver cannot read the TSC at the instant when the hardware interrupt is asserted, it approximates the interrupt time from the start timestamp the last time the ISR ran and the timer period. By default the PIT timer under Windows fires with a 10ms period. During statistical latency measurement this is reset to a 1 ms. period. Timer interrupt latency is estimated to be the maximum difference between the approximated interrupt time and the TSC value recorded in the timer ISR.

DPC latency is the delay from the time at which the software ISR enqueues a DPC (Deferred Procedure Call, see section 1.1) until the first instruction of the DPC is executed, as shown in Figure 1. Because ordinary DPCs queue in FIFO order, DPC latency encompasses the time required to enqueue and dequeue a DPC as well as the aggregate time to execute all DPCs in the DPC queue when the DPC was enqueued. For comparison with other OSes where more processing is done in the actual ISR, we also define the notion of *DPC interrupt latency*, which is simply the sum of the interrupt and DPC latencies for a single interrupt, as shown in Figure 1.

Thread latency is the delay from the time at which an ISR or DPC signals a waiting thread until the time at which the signaled thread executes the first instruction after the wait is satisfied. Thus, it measures the worst-case thread dispatch latency for a thread waiting on an interrupt, measured from the ISR or DPC itself to the first instruction executed by the thread after the wait. Thread latency encompasses a variety of thread types and priorities (e.g., kernel mode high real-time priority) and includes the time required to save and restore a thread context and obtain and/or release semaphores.

In the worst-case thread latency represents the maximum time during which the operating system disables thread scheduling. An important point to note is that thread latency subsumes thread context-switch time since, in the general case, the proper thread is not executing when an interrupt arrives.

When device drivers need to do processing at IRQL PASSIVE_LEVEL they can queue a callback function as a Work Item. Critical Work Items are dequeued and executed by the system worker thread that runs at default real-time priority (see section 1.1 for further background). Work Item latency is the delay from the time at which the Work Item is enqueued for execution until the first instruction of the Work Item routine is executed. Critical Work Item latency is comparable to thread latency for a thread running at the default real-time priority.

An additional powerful feature of statistical latency measurement is the ability to identify the cause of long latencies by sampling what was executing during the long latency periods. During statistical latency sampling, the data collector driver can generate high priority interrupts that store the instruction pointer (EIP) and register state of the interrupted task (read from the stack). When latency in a sample exceeds a user-specified threshold, these causal data samples are logged along with the latency sample to allow correlation. The interrupts are generated using the APIC timer that allows sampling periods of microsecond granularity. The sampling interval is 500 microseconds by default. If sampling interval is too large the data may not be conclusive to identify cause. If too small, the cost of instruction pointer sampling can become a significant addition to the latency.

2.2 System Analysis

The system analysis tool is used to profile various elements that contribute to the latencies identified by the statistical latency tool. The system analysis tool captures the time with interrupts held off, the duration of individual ISRs, the duration and latency of DPCs and Work Items than ran in the system during the sampling interval. It displays these events in a time-line enabling the user to visually analyze the system events that occurred between the queuing and start of execution of the users event. One model of using this feature is after using the statistical latency tool to find non-tolerable latencies under desired concurrent workloads. Then the workload causing long latencies can be repeated and the system events captured. The volume of these data points are very high so only short sampling intervals up to a few minutes are recommended in order to ease analysis.

The system analysis tool collects data on ISR execution time by hooking the IoConnectInterrupt() WDM call and registering an ISR wrapper. The start and end of each ISR is timestamped by the ISR wrapper. Many device drivers register their ISRs on bootup. The data collector driver is started early in system boot and by default hooks IoConnectInterrupt() in order to hook most of the ISRs registered in the system. Any ISR registered before the data collector can hook IoConnectInterrupt() or registered with mechanisms other than IoConnectInterrupt() will not be captured by the analyzer. Thus the tool will be unable to measure interrupt latency for that device interrupt since it does not know the time of assertion of the hardware interrupt. Although IoConnectInterrupt() is always hooked, the tool incurs the overhead of capturing data only during sampling.

When the system analysis tool is configured to collect DPC or Work Item latency and execution times, the data collector driver hooks the WDM calls to queue DPCs and Work Items. Section 3.2 has implementation details. The hooked queuing function timestamps the queue time and queues a wrapper callback function. When the system scheduler runs the DPC or Work Item, the wrapper function timestamps the start and end of execution of the DPC or Work Item callback function. The latency is measured as the difference in the start and queue times. The execution time is measured as the difference in the end and start time. The WDM interfaces that the analyzer hooks to collect DPC and Work Item scheduling data are the following: IoQueueWorkItem(), ExQueueWorkItem(), KeSetTimer(), KeSetTimerEx(), KeCancelTimer(), KeInsertQueueDpc() and KeRemoveQueueDpc().

In Windows OS when the Interrupt Request Level (IRQL) is raised to HIGH_LEVEL (see section 1.1) the system is prevented from servicing new interrupts. This adds to the latency of all system events. The system analysis tool captures such occurrences by hooking the KfRaiseIrql() and KfLowerIrql() functions. When these functions are called to raise or lower the IRQL to or from HIGH_LEVEL those events are timestamped.

Intel Pentium microprocessors including the Pentium III and Pentium 4 can mask interrupts using the CLI instruction. Interrupts are unmasked by the STI instruction or when the EFLAGs register is restored via the POPF or IRET instructions with the interrupt flag (IF) bit enabled [7]. Device drivers use this mechanism in addition to raising IRQL in order to protect critical processing from getting preempted by interrupt level execution. Hence it is important to capture the duration of interrupts being disabled on the processor for latency cause analysis. This is implemented by configuring a model-specific performance monitoring counter of the processor to count the number of processor cycles for which interrupts are masked.

2.3 Isochronous Workload Simulator

The latency characteristics of a system are highly sensitive to the amount of operating system overhead incurred to service any other application concurrently executing on the system. On personal computer and workstation systems, the execution environment is highly dynamic and may include a variety of concurrently executing applications.

It is necessary to test the robustness of a soft device driver or real-time computations with most possible workloads but this can be difficult to implement. The Isochronous Workload Simulation feature that is referred to as the Task Tool in the Analyzer documentation can be used to emulate concurrent activity that could occur on the system in conjunction with the soft device. This allows the developer to simulate various PC platform conditions that could realistically occur and assess their impact on real-time performance. This utility simulates various workloads on the system by consuming processor resources at interrupt, DPC, and various thread priority levels. In this way, the resource consumer can emulate many different kinds of software and device driver loads.

This feature is implemented as a WDM driver that executes calibrated amounts of work at the user requested priority level. The driver schedules tasks at user specified periods using the local Advanced Programmable Interrupt Controller (APIC) periodic timer, which enables it to schedule tasks every 1ms. The APIC timer ISR runs the task tool scheduler that runs through the list of periodic tasks and decrements their period counters. The period is reset at 0 and the calibrated amount of work is run at the requested priority level for the user specified duration of the task.

3. Implementation

This section describes some aspects of the implementation of the Analyzer that may be of interest to developers of Windows drivers and applications on Intel architecture platforms. It is not recommended to use these under-the-hood solutions in applications for general deployment.

3.1 Accurate High-Resolution Timer

Most real-time computations on a personal computer require scheduling responses in the range of 4 to 100s of milliseconds [5]. To implement a periodic task tool scheduler that can emulate such workloads we required an accurate clock of at least 1ms resolution.

The first choice was to use the system timer. By default the Windows OS kernel receives a clock interrupt every 10 to 15ms. Higher clock resolution can be achieved using a Win32 facility called multimedia timers. Multimedia timers are the primary mechanism available for applications to request timely execution of code at periods specified in 1ms increments. As detailed in [9], poor interaction between multimedia timers and HAL (see section 1.1) can cause the timer response to be quite poor on some versions of the Windows Operating System. This caused the task tool scheduler itself to not get scheduled in time and it indicated a very high rate of missed deadlines when emulating periodic tasks.

The solution we chose was to use an alternate clock available on the Pentium processor. The local Advanced Programmable Interrupt Controller (APIC) contains a 32-bit programmable timer for use by the local processor. The time base is derived from the processor's bus clock, divided by a configurable value from 1 to 128. The timer can be configured to interrupt the local processor with an arbitrary interrupt vector. Programming its initial-count register starts the timer. It counts down this value and generates an interrupt when the value reaches zero. In periodic mode, the initial count is reloaded automatically and counting is repeated. We chose a divide value of 1 and calculated the number of clock ticks in a microsecond. We set the initial count to be the number of bus clock ticks to make 500 microseconds. For the interrupt vector, we chose an unused entry from the Interrupt Descriptor Table (IDT). The implementation uses IDT entry 0x32 on Windows 98 and Windows ME. On Windows NT, Windows 2000 and Windows XP IDT entry 0xFD is used to register the APIC timer ISR.

The APIC timer is also used to sample the stack state during long latency periods for statistical analysis. The APIC timer ISR runs the task tool scheduler and the causal data sampler.

There are no standard interfaces in the Windows OS to program or share the local APIC and hence the analyzer has a WDM driver to provide this functionality. This driver checks the initial APIC register states to determine if anyone else is using it and if not assumes it has sole ownership. There is some ongoing effort to standardize use of this resource so that it can be shared appropriately.

3.2 Hooking WDM functions

Profiling the operating system scheduling activities would have been easy if the Windows source code was available for instrumentation. However the Real-Time Analyzer was written without reference to the Windows source code. Hence the architecture chosen to implement this functionality was to intercept the WDM scheduling functions. This choice has an additional advantage that the Analyzer is relatively insulated from ongoing revisions and upgrades to Windows since it hooks public WDM interfaces that are less likely to change between different Windows OS releases and the same analyzer works on all the Windows OSs that support WDM (i.e. Windows 98 SE to Windows XP).

The run time hooking of the WDM functions makes use of run time function intercept technology. First the queuing functions are intercepted and instrumented with timestamps to capture the queue time. The intercepted function additionally instruments the queued object by wrapping them in an instrumentation wrapper. The instrumentation wrapper logs timestamp of when the queued object calls the registered callback function and timestamp of when the callback function completes execution.

3.3 Contiguous Interrupt Disabling

When interrupts are disabled using the Pentium processor CLI instruction all maskable hardware interrupts are masked. If masked or held off too long it is a major contributor to poor real-time response because device interrupts that need processing may be delayed. Interrupts are enabled again when the interrupt flag is set in the EFLAGs register through the STI instruction or through other instructions that restore EFLAGs like POPF, IRET, task switches etc. The P6 family of processors support a performance counter event, CYCLES_INT_MASKED, which allows determining when the number of processor cycles with interrupts masked have accumulated to a certain value. The Intel Architecture Software Developer's Manual Volume 3 [7] describes the use of these counters. Our solution uses the performance counter event to determine the start of the interrupts disabling by generating an NMI as soon as the interrupt flag is reset. The NMI handler then reads the Pentium TimeStamp Counter to get the start of interrupt disabling. The NMI handler then sets off an APIC self-interrupt. When the self-interrupt ISR runs we know interrupts have been enabled again and capture the TSC to get the end of interrupt disabling. The self-interrupt handler then reinitializes the performance counter to generate an NMI when interrupt are disabled next. Our implementation generates the self-interrupt through the local APIC's Interrupt Command Register [7].

Setting the performance counter initial value to –1 causes an interrupt immediately upon completion of the NMI ISR, leading to an infinite loop. Hence it is necessary to set the initial value to be sufficient to complete the ISR, restore the interrupted context, and have some safety threshold to eliminate reentering the same ISR. The implementation initializes the counter to -1024. This means that events of shorter than 1K cycles that occurred during the measurement are missed. The last one that overflowed the counter is caught. Even for

this event the captured duration has an error of 0 – 1024 cycles. Figure 2 shows a timeline of the disabled interrupt measurement.

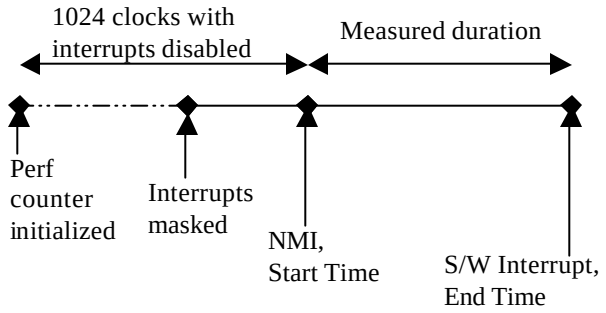

Figure 2: Timeline of Interrupt Holdoff measurement

On Pentium processors, the ability to monitor performance events and the events that can be monitored are model specific [7]. The Intel Real-Time Analyzer disables this feature of the system analysis tool on detecting the absence of CYCLES_INT_MASKED performance counter event.

4. Usage Examples

Two example usages of the Intel Real-Time Performance Analyzer are detailed in this section to illustrate the process of using it for latency analysis. The tool reports instruction addresses with their hexadecimal value. Hence a kernel debugger such as Numega* SoftIce* [14] must be used in conjunction with the analyzer to resolve addresses to symbols.

4.1 Zip Drive Latency Characterization

Parallel port zip drives used with the Windows 2000 Professional Operating System appeared to introduce significant latencies that contribute to significant deterioration of media streams. Since the Windows 2000 architecture is designed to avoid this type of problem, we analyzed what the Zip drive software was doing to cause these latencies.

The test system used a 400mHz Intel Pentium II with Intel 810 chipset system running Windows 2000 Professional. A parallel port Zip drive was installed using the Iomega* installation package ioware-w32-x86-251.exe. The system also had both the analyzer and Numega SoftICE installed. The workload was to copy large files from the system hard drive to the zip drive.

4.1.1 Step 1: Triage

The first step was to capture events in the system while these latencies are happening. To do this, we setup the analyzer to record system event latencies and statistical latencies greater than 10ms. The recording was done for a minute while a large file copy was being done to the Zip drive.

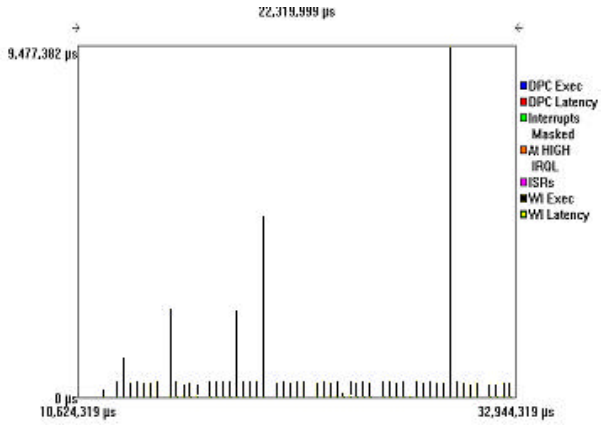

Figure 3: Detail of Timeline View

Examination of the system analysis Timeline view showed that we only detected Work Item activity, and that these Work Items appeared to have a very regular period. Figure 3 shows the timeline view of the captured events.

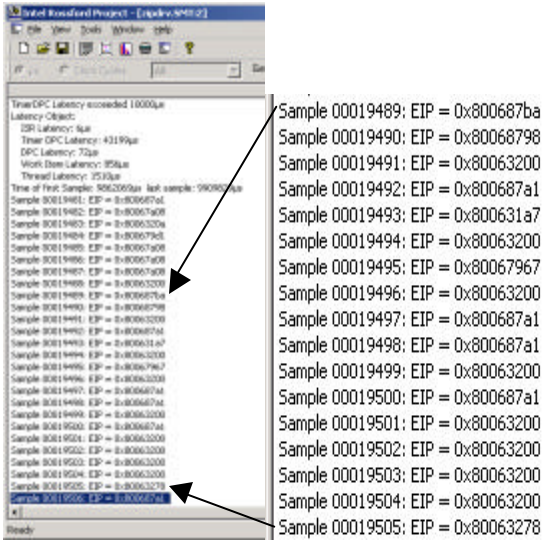

Figure 4: Causal Data View with Detail

Many statistical latency samples with latency exeeceding 10ms were recorded. Hence the causal data collected was examined. The causal sampler takes a snapshot of the instruction pointer every 500us. These addresses comprise the sample data presented in this window. Examination of these addresses can give us an indication of what was going on when the latency occurred. Notice that in this example (Figure 4), the address 0x80063200 appears several times. Using the debugger, we determined that this address belongs to the function KeStallExecutionProcessor(). This routine is used to busy wait without yielding the processor. The

presence of several calls to this routine indicated that a driver is probably operating a device in a polled mode.

At this point, we have learned that the file copy operation introduces a regular pattern of both Work Items and latencies. We also know that some driver is spending what appears to be a significant amount of time stalling instead of blocking. Next step was to determine which driver, how, and why.

4.1.2 Step 2: Identifying cause

We used the Windows DeviceManager to explore the drivers associated with devices and determined that three drivers are involved with the parallel port zip drive: parallel.sys, disk.sys, and ppa3.sys. Symbols were available for all the drivers in the DDK and they were loaded in to the debugger.

First we determined which functions were calling KeStallExecutionProcessor(). We used debugger breakpoints on this function and traced a call to it from inside ppaa3.sys. We stepped through the debugger to determine the call tree. PpaStartIo() appeared to be the last function in PPA3.sys as we unwound the call stack. Disassembly of this function showed code that uses IoQueueWorkItem() to queue a routine called PpaWorkItem(). Work Items figured heavily in our triage, so we disassembled PpaWorkItem(). It looked very straightforward:

```
PpaWorkItem:
    call IoFreeWorkItem <Work Item>
    call [_imp_KeRaiseIrqlToDpcLevel]
    call [_imp_IoStartNextPacket]
    call [_imp_KeLowerIrql]
```

Breakpointing on this routine confirmed that it was being called. Dumping the IRQL level from the debugger indicated that IRQL is raised to DISPATCH_LEVEL for the duration of the time spent in IoStartNextPacket(). This IRQL prevents thread scheduling and dequeueing of DPCs, so this Work Item really behaved as if it were a DPC. Now, we knew that ppa3.sys called KeStallExecutionProcessor() and queued Work Items.

4.1.3 Step 3: Identify how much

To identify how long the PPA3 spent in KeStallExecutionProcessor() we used the Analyzer's timeline marking feature. The analyzer supports external driver instrumentation using an event called a "marker". Drivers can call the exported function:

```
VOID RtuneInsertMarker(
    ULONGLONG       Id,
    ULONGLONG       UserVar1,
    ULONGLONG       UserVar2,
    ULONGLONG       UserVar3,
    ULONGLONG       UserVar4);
```

When called, this routine inserts a marker event into the event log with the other recorded events. The markers are then viewable in the timeline and list views. We inserted markers at the start and end of the Work Item and reported statistics regarding PPA3's use of KeStallExecutionProcessor(). To hook PpaWorkItem() it is sufficient to hook the initial call KeRaiseIrqlToDpcLevel() and last call KeLowerIrql(). Since these calls are made through import function tables the hooking was as simple as using the debugger to change the values in PPA3's import variables so that they point to our hook routines instead of the original functions.

Sequence	Started At	ID	Variable 1	Variable 2
34	350,142	0x5	0x2785	0x18b32
35	350,161	0x4	0x0	0x0
71	762,873	0x5	0x21cd	0x15202
73	762,901	0x4	0x0	0x0
111	1,183,981	0x5	0x2312	0x15eb4
113	1,184,010	0x4	0x0	0x0
150	1,614,351	0x5	0x27d2	0x18e34
152	1,614,378	0x4	0x0	0x0
191	2,020,018	0x5	0x229d	0x15a22
193	2,020,044	0x4	0x0	0x0
231	2,438,767	0x5	0x2541	0x1748a
233	2,438,795	0x4	0x0	0x0
267	2,850,937	0x5	0x214e	0x14d0c
269	2,850,962	0x4	0x0	0x0
305	3,246,081	0x5	0x2091	0x145aa
307	3,246,109	0x4	0x0	0x0
344	3,662,543	0x5	0x238c	0x16378

Figure 5: Detail of Markers View

Figure 5 shows the analyzer listing of the marker data. The entry at sequence #35 is a "begin" marker. Taking the difference between it's "Started At" time and that of the following "end" marker gives a time between markers of 412,712 us, or just over 0.4 seconds. Also note from the Variable 1 and Variable 2 entries for the end record that PPA3 made 8,653 calls to KeStallExecutionProcessor for a total stall time of 86,530us. The difference between the end of this Work Item to the start of the next is 28us.

By analyzing all the marker records and computing averages, we discovered that Work Items take 442,000us on average to complete with only 40us free between Work Items. This amounts to 99.999% CPU utilization by ppa3.sys. We also discovered that PPA3 makes an average of 10,585 calls to KeStallExecutionProcessor per Work Item and that this amounts to an average of 105,850us stall time per Work Item. In other words, of the 0.4 seconds that PPA3 spends in a Work Item, 23.9% is spent killing time waiting for the device. Thus while copy operations are taking place it is responsible for 99.99% CPU utilization and can introduce 0.4 second latencies for threads and DPCs

4.2 Soft Modem Latencies

We noticed that certain soft device implementations on Windows* NT did not work well together and with other time critical events in the system experiments were

done to investigate the soft modem processing at different scheduling priority levels. The Real-Time Analyzer was used to capture the system events when the modem is not connected, during modem training and when modem is connected. The analyzer feature of creating comparison graphs of the captured data was used to analyze the cause of the problems.

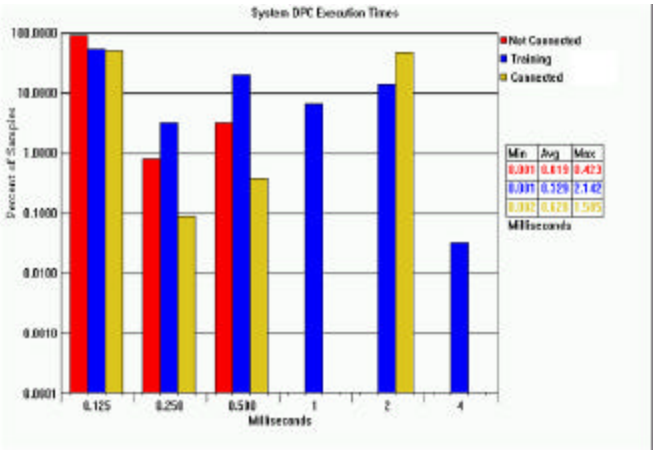

Figure 6: System DPC execution times

On a Pentium II 333 MHz system with a DPC based soft modem when the CPU utilization was less than 50% soft audio had glitches during modem training. A CD burning operation caused the modem training to fail. The analyzer graph of the DPC execution distribution in Figure 6 shows the long execution at the DPC level during modem training. This caused the audio glitches.

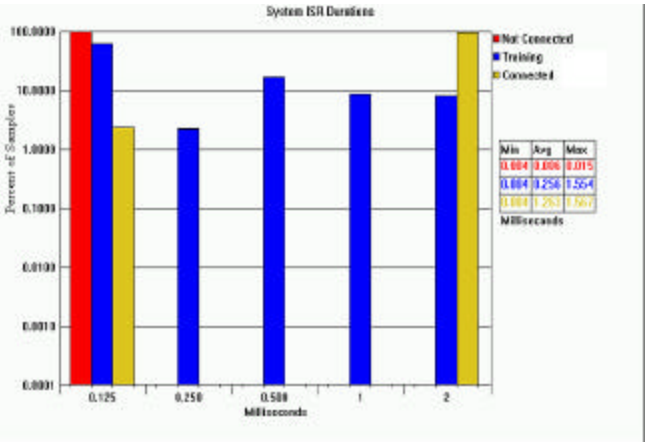

Figure 7: System ISR execution times

On a Pentium II 333 MHz system with an ISR based soft modem even though the CPU utilization was less than 50%, the soft modem caused CD burns to fail because of CD buffer underflow. SoftDVD dropped frames when the soft modem was connected. The analyzer graph of the ISR execution distribution in figure 7 shows the long ISR execution with the soft modem connected. Moving soft modem processing to ISR level fixed the modem but caused other problems.

5. Lessons Learned

Our analysis of the zip drive revealed that PPA3.SYS is not written to share system time with other drivers. The analyzer provided a framework for discovering the cause of the latencies by providing capabilities above and beyond that of periodic sampling packages. In fact, the logic flow in PPA3.SYS serves to illustrate a basic limitation with identifying causes of high CPU utilization with a periodic sampling of the instruction pointer.

In our zip drive example the reason for the calls to KeStallExecutionProcessor() turned out to be that the driver is polling a parallel port register for zip-drive status. The port is actually read in a called HAL function, READ_PORT_UCHAR. For each polling loop, the driver spends about 2 to 3 microseconds in the HAL reading the port, 10 microseconds in the KeStallExecutionProcessor() function, and no more than 0.5 microseconds in PPA3 before starting the loop again. This means that even though PPA3.SYS is responsible for the latency, there is only about a 4% chance [0.5 us / (10 + 2) us] of getting a PPA3 address in a periodic sample.

In the PC environment where large numbers of independently authored drivers and applications coexist and source code is unavailable it can be difficult to know which driver or application is using a kernel service. Periodic profiling will reveal the offending kernel service but the user (and thus the source of the problem) will often be obscured by a variety of other simultaneously active applications and drivers. The analyzer enabled us to quickly isolate a repeatable cause of the offending behavior and rapidly pin down the ultimate cause.

Our experiments with DPC and ISR based soft modem implementations lead us to conclude that independently developed device drivers and real-time applications may interfere with each other. Poorly designed applications and drivers can disrupt real-time applications. Multimedia and soft device developers should refer to the PC Design Guide [8] guidelines to carefully determine an appropriate priority. Ultimately a resource reservation mechanism will be required to completely prevent the problem.

6. Conclusion

The Intel Real-Time Performance Analyzer automates the detection of occurrences and causes of poor real-time behavior. The Analyzer is the only tool available to characterize real-time latency response of Microsoft Windows on Intel platforms. Indeed, to our knowledge it is the only tool of its kind available for any

OS. A version of the tool for Windows 98, Windows ME, Windows NT 4.0 and Windows 2000 is available for free download from http://developer.intel.com/ial/rta/. A developmental version for Windows XP and Windows NT 4.0 and XP Embedded is available on request from the authors on an "as is" basis.

The success of the Analyzer in quickly diagnosing the root cause of poor real-time performance in complex situations involving multiple independently authored real-time applications running on Windows 2000 conclusively demonstrates that OS, driver and application source code is not necessary to analyze real-time performance. As real-time software is increasingly deployed on COTS hardware and software environments lack of access to source code will become the norm. With the rise in Web-connected devices and downloaded applets it will become the norm that even on embedded systems such as cell phones the run-time application environment is dynamic and unknowable in advance. Tools like the Analyzer can help developers of real-time software to meet these challenges.

7. Acknowledgements

Thomas Barnes and Brian Belmont provided managerial support. Aric Teneyck implemented the GUI. Others who assisted include Dan Baumberger, Chandran Chellian Jaya Jeyaseelan, Sanjeev Lalgudi, Rich Minter, Dan Nowlin and Kiran Panesar.

8. References

[1] B. N. Bershad, R.P. Draves and A. Forin, "Using Microbenchmarks to Evaluate System Perfor-mance", *3rd Workshop on Workstation Operating Systems*, April, 1992.

[2] A.B. Brown and M.I. Seltzer, "Operating System Benchmarking in the Wake of Lmbench: A Case Study of the Performance of NetBSD on the Intel x86 Architecture", *1997 Sigmetrics Conference*, 1997.

[3] C. Cant, *Writing Windows WDM Device Drivers*, R&D Books, 1999.

[4] E. Cota-Robles, J. Held and T. J. Barnes, "Schedulability Analysis for Desktop Multimedia Applications: Simple Ways to Handle General-Purpose Operating Systems and Open Environments", *4th IEEE International Conference on Multimedia Computing and Systems*, June 1997.

[5] E. Cota-Robles and J. Held, "A Comparison of Windows Driver Model Latency Performance on Windows NT and Windows 98", *3rd Usenix Symposium on Operating System Design and Implementation*, Feb. 1999.

[6] Y. Endo, Z. Wang, J. B. Chen and M. I. Seltzer, " Using Latency to Evaluate Interactive System Performance", *2nd Usenix Symposium on Operating Systems Design and Implementation*, Nov. 1996.

[7] Intel Corp., *Intel Architecture Software Developer's Manual*, 3 vol., 1996-2001. Available online at http://developer.intel.com/design/Pentium4/manuals/.

[8] Intel Corp. and Microsoft Corp., *PC 99 System Design Guide*, Intel University Press, 1998.

[9] M. Jones and J. Regehr, "The Problems You're Having May Not Be the Problems You Think You're Having: Results from a Latency Study of Windows NT", *7th Workshop on Hot Topics in Operating Systems*, IEEE, March 1999.

[10] R. P. Kar, "Implementing the Rhealstone real-time benchmark", *Dr. Dobb's J.*, page 46, April 1990.

[11] L. McVoy and C. Staelin, "lmbench: Portable Tools for Performance Analysis", *1996 USENIX Technical Conf.*, January, 1996.

[12] Microsoft Corporation, "Windows 98 Driver Development Kit (DDK)" in *Microsoft Developer Network Professional Edition*, Redmond, WA, 1998.

[13] H. W. Nelson and N. I. Kameo, "Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems", *Real-Time Systems Journal*, vol. 4 (4), pages 353-383, Dec. 1992.

[14] Numega SoftIce is a kernel mode debugger for Windows. See http://www.compuware.com/products/driverstudio.

[15] W. Oney, *Programming the Microsoft Windows Driver Model*, Microsoft Press, 1999.

[16] J. K. Ousterhout, "Why Aren't Operating Systems Getting Faster As Fast as Hardware", *Usenix Summer Conference*, June, 1990.

[17] B. Shiraz, L. Welch, B. Ravindran, C. Cavanaugh and E.-N. Huh, "Dynbench: a Benchmark Suite for Dynamic Real-Time Systems", 2000.

[18] D. A. Solomon, *Inside Windows NT 2nd Edition*, Microsoft Press, Redmond, WA. 1998.

[19] M.F. Storch and J.W.S. Liu, "DRTSS: A Simulation Framework for Complex Real-Time Systems", *Real-Time Technology and Applications Symposium*, IEEE, June 1996.

[20] Ziff-Davis Corp., "Labs Notes: Benchmark 97: Inside PC Labs' Latest Tests", *PC Magazine Online*, Vol. 15, No. 21, December 3, 1996.